

TOM 4

ŁATWY PODRĘCZNIK

Jerzy Grębosz

Opus magnum C++

Misja w nadprzestrzeń C++14/17



Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki i opracowanie graficzne książki: Jerzy Grębosz

Zdjęcie Mgławicy Orzeł w grafice na okładce oraz zdjęcia łazika Curiosity i powierzchni Marsa – wykorzystane w tytułach rozdziałów – dzięki uprzejmości NASA.

Wydanie pierwsze

ISBN: 978-83-283-6587-2

Copyright © Jerzy Grębosz 2020

Printed in Poland

Helion SA

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/opc144>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przyklady/opc144.zip>

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

0	Proszę tego nie czytać!	1
0.1	Wyruszamy na kolejną wyprawę!.....	1
1	Szablony o zmiennej liczbie parametrów	3
1.1	Szablon funkcji o zmiennej liczbie parametrów (i argumentów).....	4
1.2	Jak dobrać się do argumentów tkwiących w pakiecie?.....	11
1.2.1	Ciekawe szablony zwracające rezultat	16
1.3	Szablon klas o dowolnej (zmiennej) liczbie parametrów.....	18
1.4	Trzy rodzaje pakietów parametrów szablonu	24
1.4.1	Pakiet szablonu będący pakietem wartości	25
1.5	Argumenty pakietu odbierane przez wartość, referencję, adres.....	30
1.6	Rozwinięcie według wzorca (czyli rozwinięcie „z kontekstem”).....	31
1.7	Rozwinięcie pakietu typów w klamrowej liście inicjalizatorów	34
1.7.1	Łatwe narzędzie do wypisania argumentów.....	34
1.8	Rozwinięcie pakietu na liście parametrów aktualnych innego szablonu	36
1.9	Gdzie można umieścić wyrażenia rozwijające pakiet parametrów	39
1.10	Ćwiczenia	39
2	Cechy języka wprowadzone do standardu C++14	43
2.1	Zapis dwójkowy stałych dosłownych.....	43
2.2	Separatory cyfr w stałych dosłownych	44
2.2.1	Wypisywanie liczb w postaci binarnej.....	45
2.2.2	Wczytywanie liczb dwójkowych strumieniem wejściowym.....	48
2.3	Kompilator rozpoznaje typ rezultatu funkcji	50
2.4	Deklaracja typu rezultatu <i>decltype(auto)</i>	52
2.4.1	Przykład zastosowania konstrukcji <i>decltype(auto)</i> w szablonie funkcji.....	55
2.5	Szablon definicji zmiennej	59
2.5.1	Jak to drzewiej bywało, czyli świat bez szablonów zmiennych	64
2.5.2	Teraz zobaczysz, jak prosto się to robi z C++14.....	71
2.5.3	Ciekawe zastosowanie: sprawdzenie cech charakteru danego typu	72
2.5.4	Lubię, nie lubię...	77
2.5.5	Dwa usprawnienia.....	80
2.5.6	Realizacja tego pomysłu w programie	81
2.6	Przeładowanie globalnych operatorów <i>new</i> , <i>new[]</i> , <i>delete</i> i <i>delete[]</i>	86
2.7	Nowości C++14 w wyrażeniach lambda	94
2.7.1	Przykład uogólnionego wyrażenia lambda.....	94
2.7.2	Przykład definicji obiektu na liście wychwytywania i jego inicjalizacja	96
2.7.3	Przykład wychwycenia na zasadzie przeniesienia (move)	99

2.8	C++14 a funkcje <i>constexpr</i>	104
2.8.1	Zniesienie wielu ograniczeń w ciele funkcji <i>constexpr</i>	104
2.8.2	Funkcje składowe <i>constexpr</i> w C++14 nie są już automatycznie <i>const</i>	110
2.9	Atrybuty.....	111
2.9.1	Nowy atrybut <code>[[deprecated]]</code> wprowadzony w C++14.....	112
2.9.2	Oznaczenie wybranej funkcji jako przestarzałej.....	113
2.9.3	Argument funkcji uznany za przestarzały.....	114
2.9.4	Przestarzałe niestatyczne składniki klasy: funkcja składowa i dana składowa.....	114
2.9.5	Obiekt oznaczony jako przestarzały.....	115
2.9.6	<i>deprecated</i> a zbiorcza definicja kilku zmiennych (z ewentualną inicjalizacją).....	116
2.9.7	Typy, które uznajemy za przestarzałe.....	116
2.9.8	Przestarzałe synonimy typów (w instrukcjach <i>typedef</i> i <i>using</i>).....	118
2.9.9	Oznaczenie atrybutem <code>[[deprecated]]</code> specjalizacji szablonu klasy.....	118
2.9.10	Oznaczenie atrybutem <code>[[deprecated]]</code> specjalizacji szablonu funkcji.....	119
2.10	Przewrotu nie było.....	119
2.11	Ćwiczenia.....	119

3 Cechy języka wprowadzone do standardu C++17 123

3.1	Specyfikacja wyjątków staje się częścią typu funkcji.....	123
3.2	Technika „pomijanie kopiowania” bywa teraz obowiązkiem kompilatora.....	128
3.3	Przydomek <i>alignas</i> a operatory <i>new</i> i <i>delete</i>	134
3.3.1	Przeładowanie globalnych <i>new</i> i <i>delete</i> uwzględniające wyrównanie adresów.....	135
3.3.2	Jak przeładować wyrównujące operatory <i>new/delete</i> na użytek wybranej klasy.....	141
3.4	Porządek obliczania składników w złożonych wyrażeniach – nareszcie ustalony.....	144
3.5	Stała znakowa typu <i>u8</i>	146
3.6	Szesnastkowy zapis liczb zmiennoprzecinkowych.....	147
3.6.1	Wypisywanie i wczytywanie zmiennoprzecinkowych liczb szesnastkowych.....	149
3.7	Wyrażenia poskładane w harmonijkę – ułatwienie pracy z pakietem argumentów.....	152
3.7.1	Pierwszy przykład użycia wyrażenia harmonijkowego w szablonie.....	152
3.7.2	Harmonijka z dodatkowym wyrażeniem początkowym.....	155
3.7.3	Cztery formy wyrażenia harmonijkowego.....	157
3.7.4	Kontekst wyrażenia harmonijkowego – przykład.....	159
3.8	Dozwolone słowo <i>auto</i> w deklaracji <i>template <auto></i>	161
3.9	Kompilator rozpoznaje typ parametrów klasy szablonej.....	166
3.9.1	Wektory czego innego niż widać.....	170
3.10	Instrukcja <i>if constexpr</i> – prawie jak kompilacja warunkowa.....	173
3.10.1	Instrukcja <i>if constexpr</i> rozwiązuje problem „lubianych” i „nielubianych” typów.....	176
3.11	Wyrażenia inicjalizujące w instrukcjach <i>if</i> i <i>switch</i>	179
3.12	Dowiązania strukturalne, czyli łatwe „sięganie do składników”.....	182
3.12.1	Dowiązanie do tablic zbudowanych na bazie klasy <i>std::array</i>	184
3.12.2	Łatwe sięganie do składników struktur/klas.....	185
3.12.3	Przystosowanie naszej klasy do obsługi deklaracji dowiązań.....	192
3.12.4	Przystosowanie cudzej klasy do obsługi deklaracji dowiązań.....	198
3.13	Operator preprocesora zwany <i>__has_include</i>	201
3.14	Nowe atrybuty: <i>maybe_unused</i> , <i>fallthrough</i> i <i>nodiscard</i>	203
3.14.1	Atrybut <code>[[maybe_unused]]</code>	205
3.14.2	Atrybut <code>[[fallthrough]]</code> używany w instrukcji <i>switch</i>	208
3.14.3	Atrybut <code>[[nodiscard]]</code> – nie zlekceważ mnie.....	210
3.15	Typ <i>std::byte</i> do operacji na surowych blokach pamięci.....	213
3.16	Modyfikacje istniejących cech języka.....	222
3.17	Rozluźnienie zasady inicjalizowania typów wyliczeniowych.....	223
3.18	Modyfikacja deklaracji <i>static_assert</i>	224
3.19	Prostszy sposób zapisu zagnieżdżonych przestrzeni nazw.....	225
3.20	Dozwolone słowo <i>typename</i> w parametrze określającym inny szablon.....	229

3.21	Dla zakresowej pętli <i>for</i> funkcje <i>begin</i> i <i>end</i> mogą zwracać odmienne typy	233
3.22	Rozwinięcie pakietu możliwe nawet w deklaracji <i>using</i>	239
3.23	Nowe zasady <i>auto</i> -rozpoznawania typu obiektów mających inicjalizację klamrową	245
3.24	W C++17 agregat może być nawet klasą pochodną	247
3.25	Zmiana typu rezultatu funkcji <i>std::uncaught_exception</i>	250
3.26	Ćwiczenia	254

4 Posłowie – czyli C++20 *ante portas*.....264

Skorowidz265



0 Proszę tego nie czytać!

0.1 Wyruszamy na kolejną wyprawę!

Ta książka jest kontynuacją książki *Opus magnum C++11*.

Jeśli przeczytałeś *Opus*, to znaczy, że jesteśmy już od dawna przyjaciółmi. Nie muszę się więc przedstawiać, po prostu powiem:

Witaj, stary przyjacielu! Tak się cieszę, że znowu do mnie zajrzałeś. W tej kolejnej książce chciałbym Cię zabrać w te rejony C++, o których *Opus* jeszcze nie mówił. Opowiem Ci tutaj o niezwykle ciekawych zmianach w języku C++ wprowadzonych przez standardy zwane potocznie **C++14** i **C++17**.

W rozmowach z Czytelnikami często pytam ich z obawą, czy aby *Opus* nie przytłacza ich swoją obszernością. To w końcu około 1600 stron. Na to oni odpowiadają, żebym się tym zupełnie nie przejmował, bo dla nich to czysta przyjemność. Gdyby nawet książka miała 500 stron więcej, dla nich byłoby to po prostu 500 stron więcej niezwyklej intelektualnej przygody.

Zastanawiam się, czy oni mnie czasem nie wkręcają... Z powodu tych obaw na wszelki wypadek pominąłem kiedyś w *Opusie* zagadnienie *szablonów o zmiennej (dowolnej) liczbie parametrów*. Po prostu wtedy wydało mi się to zbyt szczegółowe. Gdy w rozdziale o szablonach napisałem: „Kończę ten rozdział z poczuciem, że nie wyczerpuje on całości rozległego zagadnienia” – miałem wówczas na myśli właśnie te szablony.

Co się odwlecze...

No i może do dziś nikt by się nie zorientował, że tę sprawę pominąłem, ale w standardzie C++17 pojawiły się tak zwane wyrażenia harmonijkowe (*fold expressions*). U ich podłoża leżą właśnie te opuszczone przeze mnie szablony o zmiennej liczbie parametrów. Nie ma rady, muszę więc teraz to nadrobić. Tę książkę rozpoczyna wobec tego krótki, ale ciekawy rozdział na ten temat. Teraz myślę, że nawet dobrze się stało, bo obecnie – gdy masz już pewien dystans czasowy do *Opusu* – zagadnienie to wyda Ci się łatwe.

Po tym rozdziale następuje rozdział przedstawiający nowości języka wprowadzone przez standard C++14. W kolejnym – opis nowych zagadnień wprowadzonych przez standard C++17.

Internetowe wsparcie

Zapewne niejedno da się w tej książce poprawić. Jeśli będziesz miał jakieś uwagi, to proszę, przyślij mi je pocztą elektroniczną na adres:

jerzy.grebosz@ifj.edu.pl

Z góry wyrażam moją wdzięczność za nawet najdrobniejsze uwagi. Proszę, napisz nawet o zauważonych błędach literowych czy o swoich odczuciach, że jakiś fragment jest niejasny albo stanowczo za trudny.

Kod źródłowy przykładowych programów z tej książki można pobrać ze strony WWW wydawnictwa, a także z mojej strony WWW w internecie. Obecny adres mojej strony to:

<https://www.ifj.edu.pl/private/grebosz/>

Nawet jeśli ten adres się kiedyś zmieni, łatwo będzie znaleźć nową lokalizację za pomocą internetowej wyszukiwarki, podając hasło *Misja w nadprzestrzeń C++14/17* lub moje nazwisko.

Na mojej stronie WWW możesz też szukać odpowiedzi do ćwiczeń, które są zamieszczone na końcu poszczególnych rozdziałów tej książki. Na stronach WWW zawierających materiały uzupełniające do tej książki znajdziesz też ewentualną erratę tworzoną na bieżąco z uwag nadsyłanych przez Czytelników.

A zatem wyruszamy razem w kolejną wielką przygodę, do gwiazd. *Semper in altum!*



2.7 Nowości C++14 w wyrażeniach lambda

O wyrażeniach lambda rozmawialiśmy w *Opus magnum* w rozdziale 30. Teraz zapoznamy się z dwoma ciekawymi cechami wprowadzonymi przez standard C++14. Są to:

- 1) uogólnione wyrażenia lambda,
- 2) definicja (na liście wychwytywania) lokalnego obiektu, mającego swoją inicjalizację.

Zanim zobaczymy te cechy w przykładowych programach, kilka zdań wyjaśnienia.

2.7.1 Przykład uogólnionego wyrażenia lambda

Mówiąc najprościej, w C++11 argumenty formalne wyrażenia lambda musiały być ściśle określonego typu. Tymczasem...



Standard C++14 pozwala, by typ argumentu formalnego naszego wyrażenia lambda był parametrem. Robimy to, określając typ danego parametru słowem `auto`. Dzięki temu nasze wyrażenie lambda staje się jakby szablonem funkcji lambda o danej nazwie.

W poniższym programie zobaczysz, jakie to proste.



```
#include <iostream>
using namespace std;
//*****
int main()
{
    // wytworzenie zwykłego wyrażenia lambda C++11
    auto czy_mniejsze_int = [] (int m, int k) { return m < k; };           ❶

    cout << boolalpha;                                                    ❷
    cout << "Czy (5 < 2) ? " << czy_mniejsze_int(5, 2) << endl;          ❸
    cout << "Czy (5.1 < 5.9) ? " << czy_mniejsze_int(5.1, 5.9) << endl; ❹ // niestety!

    // C++14
    // wytworzenie uogólnionego wyrażenia lambda
    auto czy_mniejsze_uniwersalne = [] (auto m, auto k) { return m < k; }; ❺

    cout << "Czy (5 < 2) ? " << czy_mniejsze_int(5, 2) << endl;          ❻
    cout << "Czy (5.1 < 5.9) ? " << czy_mniejsze_int(5.1, 5.9) << endl; ❼
    // to działa nawet na znakach
    cout << "Czy ('a' < 'b') ? " << czy_mniejsze_uniwersalne('a', 'b') << endl; ❸
    cout << "Czy ('b' < 'a') ? " << czy_mniejsze_uniwersalne('b', 'a') << endl;

    cout << "Wolno nawet nawet porownac wartosci roznnych typow!";
    cout << "Czy (3 < 3.14) ? " << czy_mniejsze_uniwersalne(3, 3.14) << endl; ❹

    cout << "Znak 'p' ma kod liczbowy: " << int('p') << endl;

    cout << "Czy ('p' < 111) ? " << czy_mniejsze_uniwersalne('p', 111) << endl; ❺
    cout << "Czy ('p' < 113.5) ? " << czy_mniejsze_uniwersalne('p', 113.5) << endl; ❻
}

```



Na ekranie pojawi się taki tekst:

```
Czy (5 < 2) ? false
Czy (5.1 < 5.9) ? false
Czy (5 < 2) ? false
Czy (5.1 < 5.9) ? false
Czy ('a' < 'b') ? true
Czy ('b' < 'a') ? false
Wolno nawet nawet porównac wartosci roznych typow
Czy (3 < 3.14 ) ? true
Znak 'p' ma kod liczbowy: 112
Czy ('p' < 111) ? false
Czy ('p' < 113.5) ? true
```

3
4
6
7

9

10
11



Najpierw przypomnienie

Jak pamiętamy z C++11, wyrażenie lambda najczęściej wpisujemy wprost w potrzebnej nam instrukcji. Zostaje ono użyte przez tę instrukcję, a potem staje się niepotrzebne. Gdybyśmy jednak chcieli skorzystać z danego wyrażenia lambda kilkakrotnie (w kilku innych instrukcjach), możemy to zrobić, nadając mu nazwę. W praktyce nadanie nazwy polega na tym, że definiujemy „obiekt lambda” o określonej nazwie i inicjalizujemy go wymyślonym przez siebie wyrażeniem lambda.

```
auto nazwa_obiektu_lambda = wyrażenie_lambda;
```

Abyśmy się nie musieli głowić, jakiego typu jest nasze wyrażenie lambda, stosujemy zastępcze słowo kluczowe `auto`. Nie będę rozwijał tego zagadnienia, bo o tej sprawie rozmawialiśmy w *Opusie* (§30.5.1). Koniec przypomnienia, teraz o nowościach.

- 1 W naszym programie widzimy tak właśnie zdefiniowane wyrażenie lambda. Obiekt lambda, w którym będziemy je przechowywać, nazywamy długą, ale obrazową nazwą `czy_mniejsze_int`. Patrząc na ciało tego wyrażenia lambda, łatwo zauważysz, że otrzymuje ono dwa argumenty (dwie wartości typu `int`) i sprawdza, czy pierwsza wartość jest mniejsza od drugiej. Jeśli „tak”, to zwraca ono wartość `true`, jeśli „nie”, wówczas zwraca wartość `false`.

- 3 Oto wywołanie tego wyrażenia lambda dla dwóch wartości typu `int`: 5 oraz 2. Na ekranie pojawia się odpowiedź `false`.

Odpowiedź jest „s³owna” dzięki manipulatorowi `boolalpha`, którym powiedzieliśmy strumieniowi `cout`, że wolimy „s³ownie” 2.

A co będzie, jeśli to samo wyrażenie lambda zastosujemy wobec dwóch wartości typu `double`?

- 4 Oto taka sytuacja. Niestety nie zadziała to poprawnie. Skoro wartości 5.1 oraz 5.9 (czyli typu `double`) zostały wysłane wyrażeniu lambda obsługującemu wartości typu `int`, to nastąpiło ich obcięcie do typu `int`. Zatem wyrażenie lambda porównało, czy $(5 < 5)$. Wartością tego wyrażenia jest `false`, bo przecież 5 nie jest mniejsze od 5.

Wniosek? Nasze wyrażenie lambda nie nadaje się do porównywania wartości typu `double`. Nie jest uniwersalne.

Ten kłopot rozwiązuje możliwość wytworzenia uogólnionego wyrażenia lambda, na które pozwala nam standard C++14.

- 5 Oto jego definicja. Zacytujmy:

```
auto czy_mniejsze_uniwersalne = [] (auto m, auto k) { return m < k; };
```

Jak widać, zamiast określać typ argumentów `m i k`, postawiliśmy tam słowa `auto`. Dzięki temu stworzyliśmy jakby szablon wyrażenia lambda o danej nazwie. Takie uogólnione wyrażenie lambda można zastosować do wartości różnych innych typów. (Oczywiście takich, wobec których operator `<` ma sens). Oto kilka takich sytuacji.

- ❶ Użycie wyrażenia lambda `czy_mniejsze_uniwersalne` wobec argumentów typu `(int, int)`. Działa, podobnie jak to poprzednie (❸).
- ❷ Użycie wobec argumentów typu `(double, double)`. Działa poprawnie, czego poprzednia lambda (❹) nie potrafiła.
- ❸ Użycie wobec argumentów typu `(char, char)`. Także działa poprawnie.
- ❹ Dwa argumenty naszego uogólnionego wyrażenia lambda wcale nie muszą być tego samego typu. Przecież każdy z nich ma swoje własne określenie `auto`. Oto użycie tego wyrażenia wobec pary argumentów `(int, double)`. Jak widać, także i ono działa poprawnie.
- ❺ A oto bardziej karkołomne zastosowanie. Porównanie argumentów `(char, int)`, a potem (❻) argumentów `(char, double)`. (Patrząc na ekran i sprawdzając, czy i jak to działa, pamiętaj że kod ASCII znaku 'p' to 112).



Pora na podsumowanie: uogólnione wyrażenie lambda to narzędzie bardzo wygodne i bardzo proste w użyciu. Sprawia, że wyrażenie lambda, które właśnie tworzymy, może być bardziej uniwersalne, bo można je zastosować do wielu różnych typów argumentów.

❷ Dociekliwym wyjaśniam, że cały mechanizm uogólnionego wyrażenia lambda polega na tym, że kompilator zamienia je nie na obiekt funkcyjny mający funkcję składową `operator()`, ale na obiekt funkcyjny mający *szablon funkcji składowej* `operator()`. W zależności od tego, jakie podajemy argumenty aktualne naszemu wyrażeniu lambda, kompilator produkuje taką lub inną przeładowaną wersję tego operatora().

Jak widać, cały spryt uogólnionego wyrażenia lambda wynika z tego, że używa ono szablonu. Szablony zaś, jak to już kiedyś ustaliliśmy, to nie:

- ❖ *run-time* polimorfizm,
innymi słowami, wielopostaciowość zrealizowana już w trakcie pracy programu (za pomocą funkcji wirtualnej),
- ❖ *lecz compile-time* polimorfizm
inaczej: wielopostaciowość zrealizowana w trakcie kompilacji (za pomocą szablonu).

Przyznam się, że nie lubię tych uogólnionych wyrażenia lambda nazywać polimorficznymi. Niepotrzebnie kojarzy się to z funkcjami wirtualnymi, a co tu dużo mówić, ten mechanizm nie dorasta funkcjom wirtualnym do pięt.

2.7.2 Przykład definicji obiektu na liście wychwytywania i jego inicjalizacja

Jak wiadomo, wyrażenie lambda może wychwycić jakieś lokalne obiekty automatycznie dostępne w zakresie, w którym ono nastąpiło. Dzięki temu może z nich korzystać w swoim ciele.



C++14 daje nam dodatkową możliwość:

Na liście wychwytywania możemy zdefiniować jakiś dodatkowy obiekt, który nam się przyda w ciele wyrażenia lambda. Od razu inicjalizujemy go jakimś wyrażeniem.

Mówiąc ściślej – jeśli na liście wychwytywania umieścimy nazwę z inicjalizacją:

nazwa = wyrażenie_inicjalizacyjne;

to tak, jakbyśmy zdefiniowali zmienną i zainicjalizowali ją. Jej zakres to zakres ciała wyrażenia lambda. Jej typ jest taki, jakby przed jej nazwą stało słowo auto (czyli wynika on z typu wyrażenia inicjalizacyjnego).

Co ciekawe i bardzo wygodne – w wyrażeniu, które inicjalizuje zmienną, wolno nam nawet wykorzystać nazwy lokalnych (automatycznych) obiektów dostępnych w zakresie, w którym nasze wyrażenie lambda tworzymy.

Jeśli potrafisz sobie wyobrazić wyrażenie lambda jako obiekt funkcyjny, to przedstawiony tu mechanizm możesz rozumieć jako możliwość dodania do wyrażenia lambda nowych danych składowych.

Te i dalsze interesujące szczegóły zobaczymy w programie poniżej.



```
#include <iostream>
#include <memory>
using namespace std;
//*****
int main()
{
    int h = 1;
    int k = 12;
    double e = 2.71;

    // definicja wyrażenia lambda
    auto impakt = [obj] = 5, h = h + 400, zmienna = k * (k + 1), &ref = e]()
    {
        // k = 1;      błąd, bo k nie było wychwycone (służy jedynie do inicjalizacji)
        // e = 1.0;    błąd (jak wyżej)
        ref = ref + 10;
        // zmienna++;      modyfikacja dozwolona tylko wtedy, jeśli lambda...
        //                                     ...ma przydomek mutable

        cout << "obj " << obj
            << ", h = " << h
            << ", zmienna = " << zmienna
            << ", ref = " << ref
            << endl;
        return;
    };

    impakt();      // wykonanie wyrażenia lambda
    cout << "Po wykonaniu lambdy impakt h = " << h << endl;
}
```



Tak wyglądał będzie ekran po wykonaniu tego programu:

obj 5, h = 401, zmienna = 156, ref = 12.71
Po wykonaniu lambda impact h = 1



Omówienie

- ② W naszym programie definiujemy wyrażenie lambda. Definiujemy je (dla uproszczenia) nie w wywołaniu jakiejś funkcji algorytmu, ale – że tak powiem – w postaci „wolno stojącej”. Zostaje ono zapisane w obiekcie lambda o nazwie impact. Fakt, że tak właśnie postąpiliśmy, nie ma nic wspólnego z przedmiotem obecnego paragrafu. Po prostu dzięki temu uniknąłem konieczności definiowania funkcji-algorytmu.

Jak widzisz, lista wychwytywania (czyli to, co jest zamknięte w kwadratowym nawiasie) jest dość rozbudowana; jest w niej kilka członów oddzielonych przecinkami:

```
[obj = 5, h = h + 400, zmienna = k * (k + 1), &ref = e]
```

Jest ich aż tyle, bo chcę pokazać różne warianty nowego typu zapisu wychwytywania z inicjalizacją. Nie przerażaj się jednak, zwykle zastosujesz tylko jeden, no, może dwa takie człony. Omówimy je po kolei.

człon: `obj = 5`

Taki zapis na liście wychwytywania odpowiada jakby zapisowi:

```
auto obj = 5;           // pseudokod
```

co oznacza następujące polecenie dla kompilatora: zdefiniuj w ramach tego wyrażenia lambda lokalną zmienną o nazwie obj i inicjalizuj ją wartością wyrażenia (5). Wyrażenie to jest (jak wiadomo) typu int, więc niech zmienna obj będzie właśnie takiego typu.

człon: `h = h + 400`

Taki zapis mówi kompilatorowi, że chcielibyśmy, aby zdefiniował nam on lokalny obiekt o nazwie h i inicjalizował go wyrażeniem (h + 400). Ale uwaga: to h stojące na prawo od znaku = (czyli w wyrażeniu inicjalizacyjnym) to zupełnie inne h niż to po lewej. Dotyczy ono jakiegoś h, które istnieje i jest dostępne w zakresie, w którym akurat naszą lambda definiujemy. W tym przypadku będzie to obiekt ① o nazwie h, zdefiniowany w zakresie funkcji main.



Zapamiętaj to. Jeśli w wyrażeniu wychwytyjącym mamy nawet tę samą nazwę po jednej i drugiej stronie znaku =, np.:

```
h = h
```

owo h z lewej oznacza lokalny obiekt, który definiujemy na użytek ciała wyrażenia lambda, zaś to h z prawej jest nazwą jakiegoś obiektu, który powinien być dostępny w zakresie, w którym naszą lambda definiujemy.

Aby uniknąć pomyłek i nieporozumień, zwykle daję tym lokalnym obiektom inne nazwy, niż mają obiekty stojące po prawej. Przykład tego widzisz na następnej pozycji listy wychwytywania, czyli...

człon: `zmienna = k * (k + 1)`

Tutaj sprawa jest jasna i czytelna, więc nie trzeba tego objaśniać. Dodam więc tylko, że obiekt o nazwie k został tu tylko użyty w wyrażeniu inicjalizującym, ale to wcale nie znaczy, że został wychwycony. Zresztą zobacz, próba użycia go w ciele wyrażenia

lambda (3) wywoła błąd kompilatora. (Dlatego tę instrukcję musiałem opatrzyć znakami komentarza).

człon: `&ref = e`

A to jest definicja referencji (przezwoiska) o nazwie `ref`, która od razu dowiaduje się (dzięki inicjalizacji), że jest przezwoiskiem obiektu `e`. Sam obiekt `e` nie jest wychwycony, więc użycie `go` w ciele (4) byłoby błędem. Ponieważ jednak wytworzyliśmy właśnie referencję do tego obiektu, to można z niego korzystać za jej pomocą (5).

- 6 Przypominam, że wprawdzie wyrażenia lambda pozwalają nam wychwycić jakieś obiekty przez wartość (czyli przez kopię), ale kompilator nie pozwoli nam tych kopii modyfikować. Aby to było możliwe, powinniśmy naszą lambda opatrzyć przydomkiem `mutable` (zob. *Opus*, §30.3.5). Ponieważ akurat tutaj tego nie zrobiliśmy, musiałem instrukcję 6 ująć w komentarz.
- 7 Oto jest instrukcja, w której wypisujemy na ekranie wartości różnych obiektów dostępnych nam w ramach ciała naszego wyrażenia lambda. Na ekranie możesz zobaczyć na przykład wartość lokalnego obiektu `h` (401).
- 9 Potem, po zakończeniu definicji wyrażenia lambda i po wykonaniu `go` (8), wypisujemy wartość tego `h`, które było zdefiniowane w funkcji `main`. Jak widać, ta wartość nie została zmieniona i nadal wynosi 1.



Podsumujmy. C++14 wzbogacił możliwości listy wychwytywania wyrażeń lambda. Możemy tam wytwarzać (i inicjalizować) nowe pomocnicze obiekty, z których chcemy skorzystać w ciele wyrażenia lambda.

Inicjalizacja tych pomocniczych obiektów może być dowolnym wyrażeniem – nawet korzystającym z obiektów dostępnych w zakresie, w którym naszą lambda definiujemy. O pewnym szczególnym zastosowaniu takiej inicjalizacji porozmawiamy w następnym paragrafie.

2.7.3 Przykład wychwycenia na zasadzie przeniesienia (move)

Jak wiemy, lista wychwytywania pozwala wyrażeniu lambda uzyskać dostęp do obiektów przez wartość, czyli przez kopię (albo przez sporządzenie referencji do danego obiektu). A co będzie, jeśli interesującego nas obiektu nie da się kopiować, wolno go tylko przenieść?

To znaczy, co będzie, gdy informację zawartą w tym obiekcie można jedynie wyjąć z jednego obiektu i włożyć do drugiego? Nie może ona być obecna w obu obiektach równocześnie.

Wyrażenie lambda nie może takiego obiektu wychwycić przez wartość (kopię).

Przykładem takich niekopiowalnych obiektów są „sprytnie wskaźniki” realizowane przez biblioteczną klasę `std::unique_ptr`. (*Opus*, §27.7.1). Jeśli w programie posługujemy się nimi, to jak sprawić, żeby nasze wyrażenie lambda było zdolne wychwycić taki wskaźnik? W C++11 takiego obiektu wychwycić się nie dało. C++14 już nam to umożliwi. O tym teraz porozmawiamy.

Sprawę rozwiązuje bardzo prosto poznana w poprzednim paragrafie *inicjalizacja na liście wychwytywania*. Zobaczmy to na przykładzie. Będą w nim dwa wyrażenia

lambda. Jedno wychwyci sprytny wskaźnik do pojedynczego obiektu, a drugie – do tablicy obiektów.



```

#include <iostream>
#include <memory>
using namespace std;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct Tklasa                                     ❶
{
    int skladnik = 0;
    ~Tklasa() {
        cout << "Destruktor ~Tklasa" << endl;    ❷
    }
};
//*****
int main()
{
    cout << "Definicja sprytnego wskaźnika do pojedynczego obiektu\n";
    unique_ptr<Tklasa>  spryt { new Tklasa };    ❸

    { // lokalny zakres w celach szkoleniowych ----- ❹

        auto lambda1 = [sw = std::move(spryt) ] ()  ❺
        {
            sw->skladnik = 4;
            cout << "Wykonuje sie lambda1" << endl;  ❻
        };                                          ❼

        if(!spryt)
            cout << "A teraz obiekt spryt NIE ma już prawa własności " << endl;    ❼

        cout << "Przed wykonaniem lambda" << endl;
        lambda1();                                 // 1. wykonanie lambda    ❸
        lambda1();                                 // 2. wykonanie lambda

        cout << "--- Teraz nastąpi koniec zakresu istnienia lambda1\n";
    } // koniec lokalnego zakresu ----- ❹
    cout << "--- Skonczył się zakres istnienia lambda1\n";

    cout << "PROBY Z TABLICA" << endl;

    // sprytny wskaźnik do tablicy 5-elementowej (z rezerwacją tej tablicy)
    constexpr int rozmiar = 5;
    unique_ptr<Tklasa[]>  wsktab { new Tklasa[rozmiar] };    ❺

    { // drugi lokalny zakres (w celach szkoleniowych) ===== ❺

        auto lamTab = [stab = std::move(wsktab), rozmiar] ()  ❻
        {
            for(int i = 0 ; i < rozmiar ; ++i){
                stab[i].skladnik = i;
            }
            cout << "Pracuje lamTab\n";
        }
    }
}

```

```
};
cout << "Przed wywołaniem lamTab\n";
lamTab();
lamTab();
cout << "=== Zakonczył sie obszar istnienia lamTab" << endl;
} // koniec lokalnego zakresu =====
cout << "=== Koniec funkcji main" << endl;
}
```



Na ekranie pojawi się tekst:

```
Definicja sprytnego wskaźnika do pojedynczego obiektu
A teraz obiekt spryt NIE ma już prawa własności
Przed wykonaniem lambdy
Wykonuje się lambda 1
Wykonuje się lambda 1
--- Teraz nastąpi koniec zakresu istnienia lambdy 1
Destruktor ~Tklasa
--- Skonczył się zakres istnienia lambdy 1
PROBY Z TABLICA
Przed wywołaniem lamTab
Pracuje lamTab
Pracuje lamTab
=== Zakonczył sie obszar istnienia lamTab
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
=== Koniec funkcji main
```



Omówienie sposobów wychwytywania na zasadzie przenoszenia

O bibliotecznej klasie „sprytnych wskaźników” `unique_ptr` rozmawialiśmy w *Opusie* (§27.7.1). Służy ona do tworzenia obiektów zachowujących się podobnie jak wskaźniki. W sprytnym wskaźniku możesz przechować adres jakiegoś obiektu, który wytworzyłeś w zapasie pamięci operatorem `new`. Sprytny wskaźnik ma tę dodatkową zdolność, że gdy kończy się czas jego życia, to – że tak powiem – „pociąga on za sobą do grobu” obiekt, którego adres przechowywał. Jak sprytny wskaźnik to robi? Po prostu w jego destruktorze użyta jest instrukcja `delete` kasująca obiekt, którym się do tej pory opiekował.

Jak wiemy, może istnieć wiele wskaźników wskazujących na ten sam obiekt. W przypadku sprytnych wskaźników mogłoby to być groźne. Wyobraź sobie, że na jeden obiekt stworzony w zapasie pamięci operatorem `new` wskazuje kilka sprytnych wskaźników. Gdy jednemu z tych sprytnych wskaźników skończy się czas życia, jego destruktor zlikwiduje cenny obiekt w zapasie pamięci. Jeśli pozostałe wskaźniki nadal będą chciały z tym „cennym” obiektem pracować – wywoła to katastrofę.

Aby tej katastrofy uniknąć, sprytnie wskaźniki `unique_ptr` stosują zasadę, że tylko jeden z nich może w danej chwili znać adres naszego obiektu z zapasu pamięci. Jeśli zrobimy przypisanie takiego wskaźnika do jakiegoś innego:

```
spryciarz_nowy = spryciarz_stary;
```

odbędzie się to nie na zasadzie *skopiowania* cennego adresu, ale na zasadzie *przeniesienia* go. Innymi słowy, `spryciarz_stary` przekaże cenny adres `spryciarzowi_nowemu`, a sam o tym adresie zapomni.

Niby to genialne, ale w przypadku, gdybyśmy chcieli pracować z takim obiektem w ramach wyrażenia lambda, to jak zorganizować jego wychwycenie? Przecież nie można tego zrobić na zasadzie wychwycenia przez wartość (czyli przez kopię). Zatem jak taki sprytny wskaźnik może zostać wychwycony przez wyrażenie lambda?

Jak wychwycić: „sprytny wskaźnik opiekujący się **pojedynczym obiektem**”

③ `unique_ptr<Tklasa> spryt { new Tklasa };`

To jest definicja sprytnego wskaźnika o nazwie `spryt`. Jest on typu `unique_ptr<Tklasa>`, czyli może się on opiekować adresem obiektu typu `Tklasa`. (To taka pomocnicza klasa zdefiniowana na górze programu ①). W nawiasie klamrowym widzimy, że nasz sprytny wskaźnik jest od razu inicjalizowany. Czym? Adresem obiektu typu `Tklasa`, który wytwarzamy operatorem `new` w zapasie pamięci.

④ i ⑨ W celach szkoleniowych definiujemy sobie w programie pewien lokalny zakres. Oczywiście w Twoim programie nie musisz tego robić. Mnie jest on potrzebny, by pokazać Ci, kiedy dokładnie nastąpi „pociągnięcie do grobu”.

⑤ Oto definicja prostego wyrażenia lambda. Jego ciało jest rozpisane na kilka linijek.

Dla uproszczenia naszej rozmowy ta definicja nie jest zapakowana do wywołania jakiejś funkcji-algorytmu, ale jest umieszczona w obiekcie lambda o nazwie `lambda1`. (Podobnie robiliśmy w poprzednich przykładach, aby sobie dodatkowo nie zaprzętać algorytmami).



Zwróć uwagę na listę wychwytywania. Tu jest kwintesencja tego programu.

```
auto lambda1 = [sw = std::move(spryt)] () // itd.
```

Definiujemy tutaj jakiś obiekt o nazwie `sw` i inicjalizujemy go wartością wyrażenia `move(spryt)`. Innymi słowy, sprawiamy tutaj, że adres przechowywany w obiekcie `spryt` zostaje przeniesiony do obiektu `sw`.

O funkcji `move` rozmawialiśmy w Opusie w §22.10.

Pytanko: jakiego typu jest obiekt `sw`? Odpowiedź: takiego, jakby przed nim stało słowo zastępcze `auto`.

```
auto sw = std::move(spryt); // czyli typu: unique_ptr<Tklasa>
```

W ten sposób sprawiliśmy, że od tej pory obiektem wytworzonym operatorem `new` (w instrukcji ③) opiekuje się wyłącznie sprytny wskaźnik `sw`.

W ciele wyrażenia `lambda1` stawiamy jakieś przykładowe instrukcje korzystające z obiektu wskazywanego przez `sw`, następnie stawiamy klamrę `}` kończąca definicję tego ciała. Potem ⑥ następuje średnik, bo to przecież koniec długiej instrukcji, która rozpoczęła się w ⑤.

Ta duża instrukcja:

1) *zdefiniowała wyrażenie lambda i...*

2) *zapamiętała je w obiekcie lambda o nazwie `lambda1`.*

⑦ Jesteśmy znowu w zakresie funkcji `main`. Dla ciekawości sprawdzamy teraz sprytny wskaźnik o nazwie `spryt`. Jeśli teraz znajduje się w nim już `nullptr`, to znaczy, że naprawdę oddał on swój cenny skarb sprytnemu wskaźnikowi `sw`. Na ekranie widzimy ⑦, że tak rzeczywiście się stało.

- 8 Oto uruchomienie naszego wyrażenia lambda. Robię to dwukrotnie, abyś zobaczył, że sprytny wskaźnik `sw` istnieje cały czas, dopóki istnieje obiekt `lambda1`. Nie znika więc po wykonaniu funkcji `lambda`. Przecież ktoś mógłby wywołać `lambda1` po raz kolejny.
- 9 Nadchodzi koniec naszego „szkoleniowego” lokalnego zakresu, w którym zdefiniowaliśmy nasz obiekt `lambda1`. Tutaj więc obiekt `lambda1` przestaje istnieć. Skoro tak, to przestanie również istnieć jego (lokalny) sprytny wskaźnik `sw`. W chwili swojej „śmierci” tenże biblioteczny sprytny wskaźnik `sw` wywołuje swój destruktory, w którym (uwierz mi na słowo) wykonywana jest instrukcja `delete` kasująca nasz obiekt `Tklasa` z zapasu pamięci. To właśnie moment „pociągnięcia do grobu”. Czy to działa? Działa, potwierdza to gadający destruktory 2, którego wypis pojawia się na ekranie 9.

Jak wychwycić: „sprytny wskaźnik opiekujący się tablicą obiektów”?

Ponieważ w zapasie pamięci najczęściej rezerwuje się nie tyle pojedyncze obiekty, co tablice takich obiektów, zobaczmy i taką sytuację.

- 10 Oto definicja sprytnego wskaźnika mogącego wskazywać na tablicę obiektów typu `Tklasa`.

```
unique_ptr<Tklasa[]> wsktab { new Tklasa[rozmiar] };
```

Jest on typu `unique_ptr<Tklasa[]>`. Ten nawias kwadratowy sprawia, że sprytny wskaźnik dowiadyuje się, że to, czym będzie się opiekować, jest tablicą, zatem w razie likwidacji ma posłużyć się operatorem `delete []` (a nie prostym `delete`).

Definiowany tutaj sprytny wskaźnik (o nazwie `wsktab`) jest od razu inicjalizowany. Co robi ta inicjalizacja? Wytwarza w zapasie pamięci (operatorem `new[]`) pięcioelementową tablicę obiektów typu `Tklasa` i jej adres przekazuje sprytnemu wskaźnikowi. Od tej pory `wsktab` wie, którą tablicą ma się opiekować.

- 11 Znowu w celach wyłącznie ilustracyjnych definiujemy lokalny zakres. Oczywiście w swoim programie nie musisz tego robić.
- 12 To jest definicja drugiego wyrażenia lambda w naszym przykładzie. Jej ciało jest rozpisane na kilka linijek. Po nim następuje średnik 13 kończący tę długą instrukcję definicji 12. Jak widzisz, to wyrażenie lambda inicjalizuje obiekt `lambda` o nazwie `lamTab`.

Dla nas najważniejsza jest tutaj lista wychwytywania. Oto jej pierwszy człon:

```
stab = std::move(wsktab) // czyli jakby: auto stab = std::move(wsktab)
```

Jest to definicja lokalnego (dla lambdy) obiektu o nazwie `stab`. Jakiego jest on typu? Takiego jak inicjalizujące go wyrażenie. Wyrażeniem inicjalizującym jest wywołanie bibliotecznej funkcji `move`. Sprawia ona, że cenny adres, przechowywany w sprytnym wskaźniku `wsktab`, zostanie stamtąd wyjęty i włożony do sprytnego wskaźnika `stab`. (Od tej pory to wskaźnik `stab` odpowiada za ewentualne „pociągnięcie do grobu” tablicy, natomiast `wsktab` zostaje z tego obowiązku zwolniony).

Aby móc w wyrażeniu lambda pracować z tą tablicą, powinniśmy znać jej rozmiar. Załatwia to drugi człon listy wychwytywania (12). Jak widać, ten rozmiar wychwytywamy po prostu przez wartość.

- 14 Oto dwukrotne wywołanie funkcji lambda `lamTab`. Robię to dwukrotnie, abyś nie pomyślał, że to zakończenie wykonywania `lamTab` spowoduje likwidację tablicy. Nie! Przecież tablica może być potrzebna w sytuacji, gdybyśmy nasze wyrażenie lambda chcieli wywołać po raz trzeci.

Likwidacja tablicy nastąpi dopiero wtedy, gdy będzie „ginał” sam obiekt lambda lamTab. A kiedy się to stanie? Już za chwilę. To właśnie w tym celu w programie utworzyłem lokalny zakres 11 ↔ 15.

- 15 Ponieważ obiekt lambda lamTab jest zdefiniowany w lokalnym zakresie, więc teraz, gdy ten zakres się kończy, obiekt lambda zostaje zlikwidowany. Na ekranie możemy zobaczyć teksty potwierdzające, że sprytny wskaźnik (będący częścią wyrażenia lambda) na chwilę przed swą „śmiercią” wywołał operator delete[] likwidujący tablicę. Dowiadujemy się o tym znowu dzięki gadającemu destruktorowi elementów tablicy. To on wypisał na ekranie stosowne informacje 15.

Podsumujmy



W tym paragrafie dowiedzieliśmy się, że wprowadzenie w C++14 nowości, jaką jest możliwość definiowania na liście wychwytywania nowych lokalnych obiektów (wraz z ich inicjalizacją), otworzyło dodatkowe drzwi. Można teraz wychwytywać obiekty na zasadzie ich przenoszenia (a nie tylko na zasadzie kopiowania).

Zwykle wystarcza nam kopiowanie. Bywają jednak obiekty, których informacji nie można kopiować; można ją tylko przenosić. Takie są na przykład sprytnie wskaźniki `unique_ptr`. Tutaj zobaczyliśmy, jak praktycznie zrealizować takie wychwytywanie metodą przenoszenia.



PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Misja w nadprzestrzeń C++14/17

W tej książce:

- Najważniejsze informacje o nowych możliwościach języka C++
- Praktyczne przykłady zastosowania konstrukcji
- Ćwiczenia utrwalające zdobytą wiedzę

C++ – mierz wysoko!

C++ to jeden z najpopularniejszych i najpotężniejszych języków programowania. Stanowi punkt wyjścia dla wielu innych języków, które odziedziczyły po nim składnię i liczne możliwości, dzięki czemu można śmiało stwierdzić, że znajomość C++ otwiera drzwi do świata nowoczesnego programowania i jest podstawą na wymagającym rynku pracy w branży informatycznej. Czasy się zmieniają, lecz to C++ jest wciąż wybierany wszędzie tam, gdzie liczą się możliwości, elastyczność, wydajność i stabilność.

Książka, którą trzymasz w rękach, to kontynuacja genialnego kompendium *Opus magnum C++11. Programowanie w języku C++*. Autor, wybitny specjalista z ogromnym doświadczeniem w międzynarodowych projektach i twórca niezwykle popularnego podręcznika *Symfonia C++*, postanowił uzupełnić swoje dzieło o zagadnienia, dla których zabrakło miejsca w poprzednich tomach. Jeśli chcesz poszerzyć wiedzę na temat szablonów oraz poznać możliwości najnowszych standardów języka C++, nie mogłeś lepiej trafić!

Rusz w kolejną misję z C++ na pokładzie!

 Helion	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS HELIONSZKOLENIA.PL	KOD KORZYŚCI Sięgnij po więcej! ▶ 
 helion.pl		ISBN 978-83-283-6587-2 
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		9 788328 365872 Cena: 49,00 zł
INFORMATYKA W NAJLEPSZYM WYDANIU		